ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

*Кафедра прикладной математики и кибернетики*

Лабораторная работа №1

Выполнил: студент группы ЗМП-41

Лёвкин И.А.

Проверил:

Фионов А.Н.

Новосибирск, 2025

# Задание

1. Самостоятельно изучить библиотеку GMP для реализации арифметики с длинными числами. Руководство и рекомендации по установке находятся в ЭИОС. По желанию студента допускается использовать другие известные ему средства реализации арифметики с длинными числами.
2. Реализовать программу генерации чисел для операций в мультипликативной группе и в циклической подгруппе G порядка . Сгенерированные числа сохранить для последующего использования в файле.
3. Реализовать исходный алгоритм Диффи-Хеллмана, алгоритм Диффи-Хеллмана в подгруппе, алгоритм MQV. Пока без хэш-функций на последнем этапе.
4. Сделать сетевые версии программ, реализующие действия пользователей А и В на разных компьютерах локальной сети.
5. Осуществить замеры времени (в виде числа процессорных циклов) при выполнении основных этапов во всех алгоритмах и провести их сопоставление.

# Ход работы

В ходе работы была реализована программа генерации чисел p, q, g для операций в мультипликативной группе и в циклической подгруппе G порядка q. Сгенерированные числа были сохранены в двух файлах для последующего использования.

Следующим шагом было осуществление замеров времени при выполнении основных этапов во всех алгоритмах и провести их сопоставление.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Протокол | Вычисление секретного ключа на сервере (циклов) | Вычисление секретного ключа на клиенте (циклов) | Вычисление финального ключа на сервере (циклов) | Вычисление финального ключа на клиенте (циклов) |
| Классический Диффи-Хеллман | 1 318 860 | 1 199 450 | 53 584 | 55 596 |
| Диффи-Хеллман в подгруппе | 980 492 | 1 156 177 | 52 496 | 58 223 |
| MQV | 2 088 654 | 2 258 682 | 123 352 | 106 051 |

# Листинг

dh\_params.h

|  |
| --- |
| #pragma once  #include <gmp.h>  #include <string>  struct DHParams {      mpz\_t p; // Prime field      mpz\_t q; // Prime order of subgroup      mpz\_t g; // Generator      DHParams()      {          mpz\_inits(p, q, g, nullptr);      }      ~DHParams()      {          mpz\_clears(p, q, g, nullptr);      }      // Restrict copying to avoid double cleanup      DHParams(const DHParams&) = delete;      DHParams& operator=(const DHParams&) = delete;      // Allow move semantics      DHParams(DHParams&& other) noexcept      {          mpz\_init\_set(p, other.p);          mpz\_init\_set(q, other.q);          mpz\_init\_set(g, other.g);      }      DHParams& operator=(DHParams&& other) noexcept      {          if (this != &other) {              mpz\_set(p, other.p);              mpz\_set(q, other.q);              mpz\_set(g, other.g);          }          return \*this;      }  };  void save\_params\_to\_file(const DHParams& params, const std::string& filename);  void load\_params\_from\_file(DHParams& params, const std::string& filename); |

dh.h

|  |
| --- |
| #pragma once  #include <gmp.h>  void generate\_private\_key(mpz\_t private\_key, const mpz\_t q);  void generate\_public\_key(mpz\_t public\_key, const mpz\_t g,      const mpz\_t private\_key, const mpz\_t p);  void compute\_shared\_secret(mpz\_t shared\_secret, const mpz\_t public\_key,      const mpz\_t private\_key, const mpz\_t p); |

generators.h

|  |
| --- |
| #pragma once  #include <gmp.h>  void find\_multiplicative\_group\_generator(mpz\_t g, const mpz\_t q, const mpz\_t p);  void find\_cyclic\_subgroup\_generator(mpz\_t g, const mpz\_t q, const mpz\_t p, unsigned long seed); |

measure.h

|  |
| --- |
| #pragma once  #include <chrono>  #include <gmp.h>  #include <rdtsc.h>  // If use\_rdtsc = true, measure processor ticks (rdtsc).  // else - time in microseconds.  template <typename Func>  double measure\_time(Func&& func, bool use\_rdtsc = true)  {      if (use\_rdtsc) {          auto start = CC();          func();          auto end = CC();          double cycles = static\_cast<double>(end - start);          return cycles;      } else {          auto start = std::chrono::high\_resolution\_clock::now();          func();          auto end = std::chrono::high\_resolution\_clock::now();          double micros = std::chrono::duration<double, std::micro>(end - start).count();          return micros;      }  } |

mqv.h

|  |
| --- |
| #pragma once  #include <gmp.h>  #include "dh\_params.h"  struct MQVKeyPair {      mpz\_t private\_key;      mpz\_t public\_key;      MQVKeyPair()      {          mpz\_inits(private\_key, public\_key, nullptr);      }      ~MQVKeyPair()      {          mpz\_clears(private\_key, public\_key, nullptr);      }      // Restrict copy operations      MQVKeyPair(const MQVKeyPair&) = delete;      MQVKeyPair& operator=(const MQVKeyPair&) = delete;      // Allow move semantics      MQVKeyPair(MQVKeyPair&& other) noexcept      {          mpz\_init\_set(private\_key, other.private\_key);          mpz\_init\_set(public\_key, other.public\_key);      }      MQVKeyPair& operator=(MQVKeyPair&& other) noexcept      {          if (this != &other) {              mpz\_set(private\_key, other.private\_key);              mpz\_set(public\_key, other.public\_key);          }          return \*this;      }  };  void generate\_mqv\_keypair(MQVKeyPair& keypair, const DHParams& params);  void compute\_mqv\_shared\_secret(mpz\_t shared\_secret,      const MQVKeyPair& static\_keypair,      const mpz\_t ephemeral\_private,      const mpz\_t ephemeral\_public\_mine,      const mpz\_t ephemeral\_public\_theirs,      const mpz\_t static\_public\_theirs,      const DHParams& params); |

prime.h

|  |
| --- |
| #pragma once  #include <gmp.h>  bool is\_prime(const mpz\_t n, int reps = 25);  void generate\_safe\_prime(mpz\_t prime, unsigned int bits,      gmp\_randstate\_t& state);  void generate\_safe\_prime\_pair(mpz\_t q, mpz\_t p, unsigned int q\_bits,      gmp\_randstate\_t& state); |

rdtsc.h

|  |
| --- |
| #pragma once  static inline unsigned int CC() \_\_attribute\_\_((always\_inline));  static inline unsigned long long rdtsc() \_\_attribute\_\_((always\_inline));  static inline unsigned int CC()  {      int a;      asm volatile("rdtsc" : "=a"(a) : : "rdx");      return a;  }  static inline unsigned long long rdtsc()  {      unsigned int tickl, tickh;      asm volatile("rdtsc" : "=a"(tickl), "=d"(tickh));      return ((unsigned long long)tickh << 32) | tickl;  } |

dh\_params.cpp

|  |
| --- |
| #include "dh\_params.h"  #include <chrono>  #include <fstream>  void save\_params\_to\_file(const DHParams& params, const std::string& filename)  {      std::ofstream file(filename);      if (!file.is\_open()) {          throw std::runtime\_error("Could not open file for writing");      }      file << mpz\_get\_str(nullptr, 16, params.p) << std::endl;      file << mpz\_get\_str(nullptr, 16, params.q) << std::endl;      file << mpz\_get\_str(nullptr, 16, params.g) << std::endl;      file.close();  }  void load\_params\_from\_file(DHParams& params, const std::string& filename)  {      std::ifstream file(filename);      if (!file.is\_open()) {          throw std::runtime\_error("Could not open file for reading");      }      std::string line;      if (!std::getline(file, line))          throw std::runtime\_error("Failed to read p");      if (mpz\_set\_str(params.p, line.c\_str(), 16) != 0)          throw std::runtime\_error("Invalid p");      if (!std::getline(file, line))          throw std::runtime\_error("Failed to read q");      if (mpz\_set\_str(params.q, line.c\_str(), 16) != 0)          throw std::runtime\_error("Invalid q");      if (!std::getline(file, line))          throw std::runtime\_error("Failed to read g");      if (mpz\_set\_str(params.g, line.c\_str(), 16) != 0)          throw std::runtime\_error("Invalid g");      file.close();  } |

dh.cpp

|  |
| --- |
| #include "dh.h"  #include <chrono>  void generate\_private\_key(mpz\_t private\_key, const mpz\_t q)  {      gmp\_randstate\_t state;      gmp\_randinit\_default(state);      auto seed = std::chrono::system\_clock::now().time\_since\_epoch().count();      gmp\_randseed\_ui(state, static\_cast<unsigned long int>(seed));      // Generate random number between 1 and q-1      mpz\_urandomm(private\_key, state, q);      mpz\_add\_ui(private\_key, private\_key, 1);      gmp\_randclear(state);  }  void generate\_public\_key(mpz\_t public\_key, const mpz\_t g,      const mpz\_t private\_key, const mpz\_t p)  {      mpz\_powm(public\_key, g, private\_key, p);  }  void compute\_shared\_secret(mpz\_t shared\_secret, const mpz\_t public\_key,      const mpz\_t private\_key, const mpz\_t p)  {      mpz\_powm(shared\_secret, public\_key, private\_key, p);  } |

generators.cpp

|  |
| --- |
| #include "generators.h"  void find\_multiplicative\_group\_generator(mpz\_t g, const mpz\_t q, const mpz\_t p)  {      mpz\_t temp;      mpz\_init(temp);      for (mpz\_set\_ui(g, 2); mpz\_cmp(g, p) < 0; mpz\_add\_ui(g, g, 1)) { // g^q mod p != 1          mpz\_powm(temp, g, q, p);          if (mpz\_cmp\_ui(temp, 1) != 0) { // g is a generator              break;          }      }      mpz\_clear(temp);  }  void find\_cyclic\_subgroup\_generator(mpz\_t g, const mpz\_t q, const mpz\_t p, unsigned long seed)  {      mpz\_t r, temp;      mpz\_inits(r, temp, NULL);      gmp\_randstate\_t state;      gmp\_randinit\_default(state);      gmp\_randseed\_ui(state, seed);      mpz\_sub\_ui(temp, p, 1);      mpz\_divexact(temp, temp, q); // temp = (p-1)/q      do {          mpz\_urandomm(r, state, p); // r in [0, p-1]          mpz\_add\_ui(r, r, 1); // r in [1, p-1]          mpz\_powm(g, r, temp, p); // g = r^((p-1)/q) mod p      } while (mpz\_cmp\_ui(g, 1) == 0); // Repeat if g == 1 to find a valid subgroup generator      mpz\_clears(r, temp, NULL);      gmp\_randclear(state);  } |

mqv.cpp

|  |
| --- |
| #include "mqv.h"  #include "dh.h"  void generate\_mqv\_keypair(MQVKeyPair& keypair, const DHParams& params)  {      generate\_private\_key(keypair.private\_key, params.q);      generate\_public\_key(keypair.public\_key, params.g, keypair.private\_key,          params.p);  }  void compute\_mqv\_shared\_secret(mpz\_t shared\_secret,      const MQVKeyPair& static\_keypair,      const mpz\_t ephemeral\_private,      const mpz\_t ephemeral\_public\_mine,      const mpz\_t ephemeral\_public\_theirs,      const mpz\_t static\_public\_theirs,      const DHParams& params)  {      mpz\_t d, e, pow2\_l, tmp, exponent, base;      mpz\_inits(d, e, pow2\_l, tmp, exponent, base, NULL);      unsigned int l = mpz\_sizeinbase(params.q, 2) / 2;      mpz\_ui\_pow\_ui(pow2\_l, 2, l);      // X = ephemeral\_public\_mine      // d = 2^l + (X mod 2^l)      mpz\_mod(tmp, ephemeral\_public\_mine, pow2\_l);      mpz\_add(d, pow2\_l, tmp);      // Y = ephemeral\_public\_theirs      // e = 2^l + (Y mod 2^l)      mpz\_mod(tmp, ephemeral\_public\_theirs, pow2\_l);      mpz\_add(e, pow2\_l, tmp);      // x = ephemeral\_private      // a = static\_keypair.private\_key      // exponent = (x + d\*a) mod q      mpz\_mul(tmp, d, static\_keypair.private\_key);      mpz\_add(exponent, ephemeral\_private, tmp);      mpz\_mod(exponent, exponent, params.q);      // B = static\_public\_theirs      // base = (Y \* B^e) mod p      mpz\_powm(tmp, static\_public\_theirs, e, params.p);      mpz\_mul(base, ephemeral\_public\_theirs, tmp);      mpz\_mod(base, base, params.p);      // shared\_secret = base^exponent mod p      mpz\_powm(shared\_secret, base, exponent, params.p);      mpz\_clears(d, e, pow2\_l, tmp, exponent, base, NULL);  } |

prime.cpp

|  |
| --- |
| #include "prime.h"  #include <iostream>  inline bool is\_prime(const mpz\_t n, int reps)  {      return mpz\_probab\_prime\_p(n, reps) > 0;  }  void generate\_safe\_prime(mpz\_t prime, unsigned int bits,      gmp\_randstate\_t& state)  {      if (bits < 2)          throw std::invalid\_argument("Bit size must be >= 2");      do {          mpz\_urandomb(prime, state, bits); // Generate random number with specified bit length          mpz\_nextprime(prime, prime); // Find the next prime number          mpz\_setbit(prime, bits - 1); // Ensure proper bit length      } while (!is\_prime(prime));  }  void generate\_safe\_prime\_pair(mpz\_t q, mpz\_t p, unsigned int q\_bits,      gmp\_randstate\_t& state)  {      do {          // Generate q          generate\_safe\_prime(q, q\_bits, state);          // Generate p = 2q + 1          mpz\_mul\_ui(p, q, 2);          mpz\_add\_ui(p, p, 1);      } while (!is\_prime(p));  } |

generators/main.cpp

#include <chrono>

#include <iostream>

#include "dh\_params.h"

#include "generators.h"

#include "measure.h"

#include "prime.h"

#include "print.h"

void save\_params(const mpz\_t q, const mpz\_t p, const mpz\_t g,

    const std::string& params\_path)

{

    DHParams params;

    mpz\_set(params.q, q);

    mpz\_set(params.p, p);

    mpz\_set(params.g, g);

    save\_params\_to\_file(params, params\_path);

}

void demo\_parameter\_generation(const std::string& params\_dir\_path)

{

    std::cout << "Generating parameters..." << std::endl;

    gmp\_randstate\_t state;

    gmp\_randinit\_default(state);

    auto seed = std::chrono::system\_clock::now().time\_since\_epoch().count();

    gmp\_randseed\_ui(state, seed);

    mpz\_t q, p;

    mpz\_t multiplicative\_group\_g, cyclic\_group\_g;

    mpz\_inits(q, p, multiplicative\_group\_g, cyclic\_group\_g, NULL);

    unsigned int q\_bits = 256;

    auto safe\_prime\_pairs\_cycles = measure\_time([&]() {

        generate\_safe\_prime\_pair(q, p, q\_bits, state);

    });

    gmp\_randclear(state);

    auto multiplicative\_cycles = measure\_time([&]() {

        find\_multiplicative\_group\_generator(multiplicative\_group\_g, q, p);

    });

    auto cyclic\_seed = std::chrono::system\_clock::now().time\_since\_epoch().count();

    auto cyclic\_cycles = measure\_time([&]() {

        find\_cyclic\_subgroup\_generator(cyclic\_group\_g, q, p, static\_cast<unsigned long>(cyclic\_seed));

    });

    save\_params(q, p, multiplicative\_group\_g, params\_dir\_path + "multiplicative\_params.txt");

    save\_params(q, p, cyclic\_group\_g, params\_dir\_path + "cyclic\_params.txt");

    mpz\_clears(q, p, multiplicative\_group\_g, cyclic\_group\_g, NULL);

    const int name\_width = 22;

    const int cycles\_width = 12;

    print\_performance\_table(

        "Parameter generation performance",

        { { "Safe prime generation", safe\_prime\_pairs\_cycles },

            { "Multiplicative group", multiplicative\_cycles },

            { "Cyclic group", cyclic\_cycles } },

        name\_width, cycles\_width);

}

int main()

{

    const std::string params\_dir\_path = "";

    demo\_parameter\_generation(params\_dir\_path);

    return EXIT\_SUCCESS;

}

demo/main.cpp

|  |
| --- |
| #include <rdtsc.h>  #include <chrono>  #include <iomanip>  #include <iostream>  #include <sstream>  #include <string>  #include <vector>  #include "dh.h"  #include "dh\_params.h"  #include "generators.h"  #include "measure.h"  #include "mqv.h"  #include "prime.h"  #include "print.h"  const int NAME\_WIDTH = 24;  const int CYCLES\_WIDTH = 20;  void demo\_dh(const std::string& params\_path)  {      mpz\_t alice\_private, alice\_public, bob\_private, bob\_public, alice\_secret,          bob\_secret;      mpz\_inits(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      // Keys      auto alice\_key\_time = measure\_time([&]() {          generate\_private\_key(alice\_private, params.q);          generate\_public\_key(alice\_public, params.g, alice\_private, params.p);      });      auto bob\_key\_time = measure\_time([&]() {          generate\_private\_key(bob\_private, params.q);          generate\_public\_key(bob\_public, params.g, bob\_private, params.p);      });      // Shared secrets      auto alice\_shared\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(alice\_secret, bob\_public, alice\_private, params.p);      });      auto bob\_shared\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(bob\_secret, alice\_public, bob\_private, params.p);      });      print\_performance\_table(          "Diffie-Hellman protocol",          { { "Alice key", alice\_key\_time },              { "Bob key", bob\_key\_time },              { "Alice shared secret", alice\_shared\_secret\_time },              { "Bob shared secret", bob\_shared\_secret\_time } },          NAME\_WIDTH, CYCLES\_WIDTH);      std::cout << std::endl;      print\_secrets(alice\_secret, bob\_secret);      // Cleanup      mpz\_clears(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);  }  void demo\_subgroup\_dh(const std::string& params\_path)  {      mpz\_t alice\_private, alice\_public, bob\_private, bob\_public;      mpz\_t alice\_secret, bob\_secret;      mpz\_inits(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      // Keys      auto alice\_key\_time = measure\_time([&]() {          generate\_private\_key(alice\_private, params.q);          generate\_public\_key(alice\_public, params.g, alice\_private, params.p);      });      auto bob\_key\_time = measure\_time([&]() {          generate\_private\_key(bob\_private, params.q);          generate\_public\_key(bob\_public, params.g, bob\_private, params.p);      });      // Shared secrets      auto alice\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(alice\_secret, bob\_public, alice\_private, params.p);      });      auto bob\_secret\_time = measure\_time([&]() {          compute\_shared\_secret(bob\_secret, alice\_public, bob\_private, params.p);      });      print\_performance\_table(          "Subgroup Diffie-Hellman protocol",          { { "Alice key", alice\_key\_time },              { "Bob key", bob\_key\_time },              { "Alice shared secret", alice\_secret\_time },              { "Bob shared secret", bob\_secret\_time } },          NAME\_WIDTH, CYCLES\_WIDTH);      std::cout << std::endl;      print\_secrets(alice\_secret, bob\_secret);      // Cleanup      mpz\_clears(alice\_private, alice\_public, bob\_private, bob\_public,          alice\_secret, bob\_secret, NULL);  }  void demo\_mqv(const std::string& params\_path)  {      mpz\_t alice\_ephemeral\_private, alice\_ephemeral\_public;      mpz\_t bob\_ephemeral\_private, bob\_ephemeral\_public;      mpz\_t alice\_secret, bob\_secret;      mpz\_inits(alice\_ephemeral\_private, alice\_ephemeral\_public,          bob\_ephemeral\_private, bob\_ephemeral\_public,          alice\_secret, bob\_secret, NULL);      DHParams params;      // Load parameters      load\_params\_from\_file(params, params\_path);      MQVKeyPair alice\_static, bob\_static;      // Static keys      auto alice\_static\_time = measure\_time([&]() { generate\_mqv\_keypair(alice\_static, params); });      auto bob\_static\_time = measure\_time([&]() { generate\_mqv\_keypair(bob\_static, params); });      // Ephemeral keys      auto alice\_ephemeral\_time = measure\_time([&]() {          generate\_private\_key(alice\_ephemeral\_private, params.q);          generate\_public\_key(alice\_ephemeral\_public, params.g,              alice\_ephemeral\_private, params.p);      });      auto bob\_ephemeral\_time = measure\_time([&]() {          generate\_private\_key(bob\_ephemeral\_private, params.q);          generate\_public\_key(bob\_ephemeral\_public, params.g,              bob\_ephemeral\_private, params.p);      });      // Shared secrets      auto alice\_secret\_time = measure\_time([&]() {          compute\_mqv\_shared\_secret(alice\_secret, alice\_static,              alice\_ephemeral\_private, alice\_ephemeral\_public,              bob\_ephemeral\_public, bob\_static.public\_key,              params);      });      auto bob\_secret\_time = measure\_time([&]() {          compute\_mqv\_shared\_secret(bob\_secret, bob\_static,              bob\_ephemeral\_private, bob\_ephemeral\_public,              alice\_ephemeral\_public, alice\_static.public\_key,              params);      });      print\_performance\_table(          "MQV protocol",          { { "Alice static keypair", alice\_static\_time },              { "Bob static keypair", bob\_static\_time },              { "Alice ephemeral key", alice\_ephemeral\_time },              { "Bob ephemeral key", bob\_ephemeral\_time },              { "Alice MQV shared secret", alice\_secret\_time },              { "Bob MQV shared secret", bob\_secret\_time } },          NAME\_WIDTH, CYCLES\_WIDTH);      std::cout << std::endl;      print\_secrets(alice\_secret, bob\_secret);      mpz\_clears(alice\_ephemeral\_private, alice\_ephemeral\_public,          bob\_ephemeral\_private, bob\_ephemeral\_public,          alice\_secret, bob\_secret, NULL);  }  int main()  {      const std::string params\_dir\_path = "";      const std::string multiplicative\_params\_path          = params\_dir\_path + "multiplicative\_params.txt";      const std::string cyclic\_params\_path = params\_dir\_path + "cyclic\_params.txt";      demo\_dh(multiplicative\_params\_path);      std::cout << std::endl;      demo\_subgroup\_dh(cyclic\_params\_path);      std::cout << std::endl;      demo\_mqv(cyclic\_params\_path);      return EXIT\_SUCCESS;  } |

duo\_client

cli.h

|  |
| --- |
| #pragma once  #include "protocol.h"  #include <string>  enum class RunMode {      Server,      Client  };  struct Args {      RunMode mode;      Protocol protocol;      std::string params\_path;      std::string server\_ip;  };  RunMode parse\_run\_mode(std::string mode);  Args parse\_args(int argc, char\* argv[]);  void print\_usage(std::string& path); |

client.h

|  |
| --- |
| #pragma once  #include "protocol.h"  #include <string>  void run\_client(const std::string& params\_path, const std::string& server\_ip, Protocol protocol);  void run\_dh\_client(const std::string& params\_path, const std::string& server\_ip);  void run\_mqv\_client(const std::string& params\_path, const std::string& server\_ip); |

network\_session.h

|  |
| --- |
| #pragma once  #include <iostream>  #include <stdexcept>  #include <string>  #ifdef \_WIN32  #include <winsock2.h>  #include <ws2tcpip.h>  #else  #include <arpa/inet.h>  #include <netinet/in.h>  #include <sys/socket.h>  #include <unistd.h>  #endif  #include <gmp.h>  class NetworkSession {  public:      NetworkSession();      ~NetworkSession();      void start\_server(unsigned port = DEFAULT\_PORT);      void connect\_to\_server(const std::string& server\_ip, unsigned port = DEFAULT\_PORT);      bool send\_value(const mpz\_t value);      bool receive\_value(mpz\_t value);  private:  #ifdef \_WIN32      SOCKET serverSocket;      SOCKET clientSocket;      WSADATA wsaData;  #else      int serverSocket;      int clientSocket;  #endif      static const unsigned DEFAULT\_PORT = 12345;      NetworkSession(const NetworkSession&) = delete;      NetworkSession& operator=(const NetworkSession&) = delete;  }; |

network.h

|  |
| --- |
| #pragma once  #include <gmp.h>  #include <string>  #ifdef \_WIN32  #include <winsock2.h>  #include <ws2tcpip.h>  typedef SOCKET SocketType;  #define INVALID\_SOCKET\_VALUE INVALID\_SOCKET  #define CLOSE\_SOCKET closesocket  #else  #include <arpa/inet.h>  #include <netinet/in.h>  #include <sys/socket.h>  #include <unistd.h>  typedef int SocketType;  #define INVALID\_SOCKET\_VALUE (-1)  #define CLOSE\_SOCKET close  #endif  const int BUFFER\_SIZE = 4096;  std::string mpz\_to\_string(const mpz\_t num);  void string\_to\_mpz(mpz\_t num, const std::string& str);  bool send\_mpz(SocketType sock, const mpz\_t num);  bool receive\_mpz(SocketType sock, mpz\_t num); |

protocol.h

|  |
| --- |
| #pragma once  #include <stdexcept>  enum class Protocol {      DH,      MQV  };  inline Protocol parse\_protocol(std::string str)  {      if (str == "dh") {          return Protocol::DH;      } else if (str == "mqv") {          return Protocol::MQV;      } else {          throw std::invalid\_argument("Unknown protocol: " + str);      }  } |

server.h

|  |
| --- |
| #pragma once  #include "protocol.h"  #include <string>  void run\_server(const std::string& params\_path, Protocol protocol);  void run\_dh\_server(const std::string& params\_path);  void run\_mqv\_server(const std::string& params\_path); |

cli.cpp

|  |
| --- |
| #include "cli.h"  #include <cstring>  #include <iostream>  #include <stdexcept>  RunMode parse\_run\_mode(std::string mode)  {      if (mode == "-s") {          return RunMode::Server;      } else if (mode == "-c") {          return RunMode::Client;      } else {          throw std::invalid\_argument("Unknown run mode: " + mode);      }  }  Args parse\_args(int argc, char\* argv[])  {      RunMode mode = parse\_run\_mode(argv[1]);      std::string server\_ip = "";      if (mode == RunMode::Client) {          if (argc != 5 || strlen(argv[4]) == 0) {              server\_ip = "";              throw std::invalid\_argument("Server IP is empty");          }          server\_ip = argv[4];      }      Protocol protocol = parse\_protocol(argv[2]);      std::string params\_path = argv[3];      Args args = {          mode,          protocol,          params\_path,          server\_ip,      };      return args;  }  std::string filename(std::string& path)  {      size\_t lastSlash = path.find\_last\_of("/\\");      if (lastSlash != std::string::npos) {          std::string filename = path.substr(lastSlash + 1);          return filename;      } else {          return path;      }  }  void print\_usage(std::string& path)  {      const std::string cleaned\_name = filename(path);      std::cout << "Usage:" << std::endl;      std::cout << "  Server mode: " << cleaned\_name << " -s <protocol> <params\_file>" << std::endl;      std::cout << "  Client mode: " << cleaned\_name << " -c <protocol> <params\_file> <server\_ip>" << std::endl;      std::cout << "Protocols:" << std::endl;      std::cout << "  dh         - Diffie-Hellman (subgroup depends on params)" << std::endl;      std::cout << "  mqv        - MQV protocol" << std::endl;  } |

client.cpp

|  |
| --- |
| #include "client.h"  #include "dh.h"  #include "dh\_params.h"  #include "measure.h"  #include "mqv.h"  #include "network\_session.h"  #include "print.h"  #include <iostream>  const int NAME\_WIDTH = 24;  const int CYCLES\_WIDTH = 20;  void run\_client(const std::string& params\_path, const std::string& server\_ip, Protocol protocol)  {      switch (protocol) {      case Protocol::DH:          run\_dh\_client(params\_path, server\_ip);          break;      case Protocol::MQV:          run\_mqv\_client(params\_path, server\_ip);          break;      }  }  void run\_dh\_client(const std::string& params\_path, const std::string& server\_ip)  {      NetworkSession session;      session.connect\_to\_server(server\_ip);      std::cout << "Starting Diffie-Hellman key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t client\_private, client\_public, server\_public, client\_secret;      mpz\_inits(client\_private, client\_public, server\_public, client\_secret, NULL);      try {          auto client\_key\_time = measure\_time([&]() {              generate\_private\_key(client\_private, params.q);              generate\_public\_key(client\_public, params.g, client\_private, params.p);          });          if (!session.receive\_value(server\_public)) {              throw std::runtime\_error("Failed to receive server's public key");          }          if (!session.send\_value(client\_public)) {              throw std::runtime\_error("Failed to send public key");          }          auto client\_secret\_time = measure\_time([&]() {              compute\_shared\_secret(client\_secret, server\_public, client\_private, params.p);          });          print\_performance\_table(              "Diffie-Hellman protocol (Client)",              { { "Client key", client\_key\_time },                  { "Client shared secret", client\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nClient's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, client\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(client\_private, client\_public, server\_public, client\_secret, NULL);  }  void run\_mqv\_client(const std::string& params\_path, const std::string& server\_ip)  {      NetworkSession session;      session.connect\_to\_server(server\_ip);      std::cout << "Starting MQV key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t ephemeral\_private, ephemeral\_public, server\_ephemeral\_public;      mpz\_t server\_static\_public, client\_secret;      MQVKeyPair client\_static\_keypair;      mpz\_inits(ephemeral\_private, ephemeral\_public, server\_ephemeral\_public,          server\_static\_public, client\_secret, NULL);      try {          // Generate static key pair          auto client\_static\_time = measure\_time([&]() {              generate\_mqv\_keypair(client\_static\_keypair, params);          });          // Generate ephemeral key pair          auto client\_ephemeral\_time = measure\_time([&]() {              generate\_private\_key(ephemeral\_private, params.q);              generate\_public\_key(ephemeral\_public, params.g, ephemeral\_private, params.p);          });          // Exchange static public keys          if (!session.receive\_value(server\_static\_public)) {              throw std::runtime\_error("Failed to receive server's static public key");          }          if (!session.send\_value(client\_static\_keypair.public\_key)) {              throw std::runtime\_error("Failed to send static public key");          }          // Exchange ephemeral public keys          if (!session.receive\_value(server\_ephemeral\_public)) {              throw std::runtime\_error("Failed to receive server's ephemeral public key");          }          if (!session.send\_value(ephemeral\_public)) {              throw std::runtime\_error("Failed to send ephemeral public key");          }          // Compute shared secret          auto client\_secret\_time = measure\_time([&]() {              compute\_mqv\_shared\_secret(client\_secret, client\_static\_keypair,                  ephemeral\_private, ephemeral\_public,                  server\_ephemeral\_public, server\_static\_public,                  params);          });          print\_performance\_table(              "MQV protocol (Client)",              { { "Client static key", client\_static\_time },                  { "Client ephemeral key", client\_ephemeral\_time },                  { "Client shared secret", client\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nClient's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, client\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(ephemeral\_private, ephemeral\_public, server\_ephemeral\_public,          server\_static\_public, client\_secret, NULL);  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "cli.h"  #include "client.h"  #include "server.h"  int main(int argc, char\* argv[])  {      std::string executable\_name = argv[0];      if (argc < 4) {          print\_usage(executable\_name);          return EXIT\_FAILURE;      }      Args args;      try {          args = parse\_args(argc, argv);      } catch (const std::exception& e) {          std::cerr << e.what() << std::endl;          print\_usage(executable\_name);          return EXIT\_FAILURE;      }      try {          switch (args.mode) {          case RunMode::Server:              run\_server(args.params\_path, args.protocol);              break;          case RunMode::Client:              if (args.server\_ip.empty()) {                  return EXIT\_FAILURE;              }              run\_client(args.params\_path, args.server\_ip, args.protocol);              break;          default:              print\_usage(executable\_name);              return EXIT\_FAILURE;          }      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;          return EXIT\_FAILURE;      }      return EXIT\_SUCCESS;  } |

network\_session.cpp

|  |
| --- |
| #include "network\_session.h"  #include "network.h"  NetworkSession::NetworkSession()      : serverSocket(INVALID\_SOCKET\_VALUE)      , clientSocket(INVALID\_SOCKET\_VALUE)  {  #ifdef \_WIN32      if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {          throw std::runtime\_error("WSAStartup failed");      }  #endif  }  NetworkSession::~NetworkSession()  {      if (clientSocket != INVALID\_SOCKET\_VALUE)          CLOSE\_SOCKET(clientSocket);      if (serverSocket != INVALID\_SOCKET\_VALUE)          CLOSE\_SOCKET(serverSocket);  #ifdef \_WIN32      WSACleanup();  #endif  }  void NetworkSession::start\_server(unsigned port)  {      serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);      if (serverSocket == INVALID\_SOCKET\_VALUE) {          throw std::runtime\_error("Socket creation failed");      }      int opt = 1;  #ifdef \_WIN32      if (setsockopt(serverSocket, SOL\_SOCKET, SO\_REUSEADDR,              reinterpret\_cast<const char\*>(&opt), sizeof(opt))          < 0) {  #else      if (setsockopt(serverSocket, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)) < 0) {  #endif          CLOSE\_SOCKET(serverSocket);          throw std::runtime\_error("Setsockopt failed");      }      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      serverAddr.sin\_addr.s\_addr = INADDR\_ANY;      serverAddr.sin\_port = htons(port);      if (bind(serverSocket, (struct sockaddr\*)&serverAddr, sizeof(serverAddr)) < 0) {          CLOSE\_SOCKET(serverSocket);          throw std::runtime\_error("Bind failed");      }      if (listen(serverSocket, 1) < 0) {          CLOSE\_SOCKET(serverSocket);          throw std::runtime\_error("Listen failed");      }      std::cout << "Server is listening on port " << port << std::endl;      clientSocket = accept(serverSocket, NULL, NULL);      if (clientSocket == INVALID\_SOCKET\_VALUE) {          CLOSE\_SOCKET(serverSocket);          throw std::runtime\_error("Accept failed");      }      std::cout << "Client connected." << std::endl;  }  void NetworkSession::connect\_to\_server(const std::string& server\_ip, unsigned port)  {      clientSocket = socket(AF\_INET, SOCK\_STREAM, 0);      if (clientSocket == INVALID\_SOCKET\_VALUE) {          throw std::runtime\_error("Socket creation failed");      }      sockaddr\_in serverAddr;      serverAddr.sin\_family = AF\_INET;      serverAddr.sin\_port = htons(port);      if (inet\_pton(AF\_INET, server\_ip.c\_str(), &serverAddr.sin\_addr) <= 0) {          CLOSE\_SOCKET(clientSocket);          throw std::runtime\_error("Invalid address");      }      if (connect(clientSocket, (struct sockaddr\*)&serverAddr, sizeof(serverAddr)) < 0) {          CLOSE\_SOCKET(clientSocket);          throw std::runtime\_error("Connection failed");      }      std::cout << "Connected to server." << std::endl;  }  bool NetworkSession::send\_value(const mpz\_t value)  {      return send\_mpz(clientSocket, value);  }  bool NetworkSession::receive\_value(mpz\_t value)  {      return receive\_mpz(clientSocket, value);  } |

network.cpp

|  |
| --- |
| #include "network.h"  #include <cstring>  #include <iostream>  std::string mpz\_to\_string(const mpz\_t num)  {      char\* str = mpz\_get\_str(nullptr, 16, num);      std::string result(str);      free(str);      return result;  }  void string\_to\_mpz(mpz\_t num, const std::string& str)  {      mpz\_set\_str(num, str.c\_str(), 16);  }  bool send\_mpz(SocketType sock, const mpz\_t num)  {      std::string str = mpz\_to\_string(num);      str += "\n";  #ifdef \_WIN32      int result = send(sock, str.c\_str(), static\_cast<int>(str.length()), 0);      return result != SOCKET\_ERROR && static\_cast<size\_t>(result) == str.length();  #else      ssize\_t bytes\_sent = send(sock, str.c\_str(), str.length(), 0);      return bytes\_sent != -1 && static\_cast<size\_t>(bytes\_sent) == str.length();  #endif  }  bool receive\_mpz(SocketType sock, mpz\_t num)  {      char buffer[BUFFER\_SIZE];      std::string str;  #ifdef \_WIN32      int bytes;      while ((bytes = recv(sock, buffer, BUFFER\_SIZE - 1, 0)) > 0) {  #else      ssize\_t bytes;      while ((bytes = recv(sock, buffer, BUFFER\_SIZE - 1, 0)) > 0) {  #endif          buffer[bytes] = '\0';          str += buffer;          if (str.find('\n') != std::string::npos) {              break;          }      }      if (bytes <= 0)          return false;      str = str.substr(0, str.find('\n'));      string\_to\_mpz(num, str);      return true;  } |

server.cpp

|  |
| --- |
| #include "server.h"  #include "dh.h"  #include "dh\_params.h"  #include "measure.h"  #include "mqv.h"  #include "network\_session.h"  #include "print.h"  #include <iostream>  const int NAME\_WIDTH = 24;  const int CYCLES\_WIDTH = 20;  void run\_server(const std::string& params\_path, Protocol protocol)  {      switch (protocol) {      case Protocol::DH:          run\_dh\_server(params\_path);          break;      case Protocol::MQV:          run\_mqv\_server(params\_path);          break;      }  }  void run\_dh\_server(const std::string& params\_path)  {      NetworkSession session;      session.start\_server();      std::cout << "Starting Diffie-Hellman key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t server\_private, server\_public, client\_public, server\_secret;      mpz\_inits(server\_private, server\_public, client\_public, server\_secret, NULL);      try {          auto server\_key\_time = measure\_time([&]() {              generate\_private\_key(server\_private, params.q);              generate\_public\_key(server\_public, params.g, server\_private, params.p);          });          if (!session.send\_value(server\_public)) {              throw std::runtime\_error("Failed to send public key");          }          if (!session.receive\_value(client\_public)) {              throw std::runtime\_error("Failed to receive client's public key");          }          auto server\_secret\_time = measure\_time([&]() {              compute\_shared\_secret(server\_secret, client\_public, server\_private, params.p);          });          print\_performance\_table(              "Diffie-Hellman protocol (Server)",              { { "Server key", server\_key\_time },                  { "Server shared secret", server\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nServer's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, server\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(server\_private, server\_public, client\_public, server\_secret, NULL);  }  void run\_mqv\_server(const std::string& params\_path)  {      NetworkSession session;      session.start\_server();      std::cout << "Starting MQV key exchange..." << std::endl;      DHParams params;      load\_params\_from\_file(params, params\_path);      mpz\_t ephemeral\_private, ephemeral\_public, client\_ephemeral\_public;      mpz\_t client\_static\_public, server\_secret;      MQVKeyPair server\_static\_keypair;      mpz\_inits(ephemeral\_private, ephemeral\_public, client\_ephemeral\_public,          client\_static\_public, server\_secret, NULL);      try {          // Generate static key pair          auto server\_static\_time = measure\_time([&]() {              generate\_mqv\_keypair(server\_static\_keypair, params);          });          // Generate ephemeral key pair          auto server\_ephemeral\_time = measure\_time([&]() {              generate\_private\_key(ephemeral\_private, params.q);              generate\_public\_key(ephemeral\_public, params.g, ephemeral\_private, params.p);          });          // Exchange static public keys          if (!session.send\_value(server\_static\_keypair.public\_key)) {              throw std::runtime\_error("Failed to send static public key");          }          if (!session.receive\_value(client\_static\_public)) {              throw std::runtime\_error("Failed to receive client's static public key");          }          // Exchange ephemeral public keys          if (!session.send\_value(ephemeral\_public)) {              throw std::runtime\_error("Failed to send ephemeral public key");          }          if (!session.receive\_value(client\_ephemeral\_public)) {              throw std::runtime\_error("Failed to receive client's ephemeral public key");          }          // Compute shared secret          auto server\_secret\_time = measure\_time([&]() {              compute\_mqv\_shared\_secret(server\_secret, server\_static\_keypair,                  ephemeral\_private, ephemeral\_public,                  client\_ephemeral\_public, client\_static\_public,                  params);          });          print\_performance\_table(              "MQV protocol (Server)",              { { "Server static key", server\_static\_time },                  { "Server ephemeral key", server\_ephemeral\_time },                  { "Server shared secret", server\_secret\_time } },              NAME\_WIDTH, CYCLES\_WIDTH);          std::cout << "\nServer's shared secret:" << std::endl;          mpz\_out\_str(stdout, 16, server\_secret);          std::cout << std::endl;      } catch (const std::exception& e) {          std::cerr << "Error: " << e.what() << std::endl;      }      mpz\_clears(ephemeral\_private, ephemeral\_public, client\_ephemeral\_public,          client\_static\_public, server\_secret, NULL);  } |

visual/print.h

|  |
| --- |
| #pragma once  #include <chrono>  #include <gmp.h>  #include <vector>  std::string center(const std::string s, const int w);  std::string left(const std::string s, const int w);  std::string prd(const unsigned int x, const int width);  void print\_number(const char\* label, const mpz\_t num);  void print\_secrets(mpz\_t alice\_secret, mpz\_t bob\_secret);  void print\_protocol\_header(const std::string& protocol\_name, int name\_width,      int cycles\_width);  void print\_performance\_row(const std::string& name, unsigned int cycles,      int name\_width, int cycles\_width);  void print\_performance\_table(      const std::string& protocol\_name,      const std::vector<std::tuple<std::string, unsigned int>>& performance\_data,      int name\_width, int cycles\_width); |

visual/print.cpp

|  |
| --- |
| #include "print.h"  #include <chrono>  #include <iomanip>  #include <iostream>  #include <sstream>  #include <vector>  /\*! Center-aligns string within a field of width w. Pads with blank spaces      to enforce alignment. \*/  std::string center(const std::string s, const int w)  {      std::stringstream ss, spaces;      int padding = w - s.size(); // count excess room to pad      for (int i = 0; i < padding / 2; ++i)          spaces << " ";      ss << spaces.str() << s << spaces.str(); // format with padding      if (padding > 0 && padding % 2 != 0) // if odd #, add 1 space          ss << " ";      return ss.str();  }  /\*! Left-aligns string within a field of width w. Pads with blank spaces      to enforce alignment. \*/  std::string left(const std::string s, const int w)  {      std::stringstream ss, spaces;      int padding = w - s.size(); // count excess room to pad      for (int i = 0; i < padding; ++i)          spaces << " ";      ss << s << spaces.str(); // format with padding      return ss.str();  }  /\* Convert double to string with specified number of places after the decimal     and left padding. \*/  std::string prd(const unsigned int x, const int width)  {      std::stringstream ss;      ss << std::right;      ss.fill(' '); // fill space around displayed #      ss.width(width); // set  width around displayed #      ss << x;      return ss.str();  }  void print\_number(const char\* label, const mpz\_t num)  {      std::cout << label << ": " << mpz\_get\_str(nullptr, 16, num) << std::endl;  }  void print\_secrets(mpz\_t alice\_secret, mpz\_t bob\_secret)  {      print\_number("Alice's shared secret", alice\_secret);      print\_number("  Bob's shared secret", bob\_secret);      std::cout << "Secrets match: "                << (mpz\_cmp(alice\_secret, bob\_secret) == 0 ? "Yes" : "No")                << std::endl;  }  void print\_protocol\_header(const std::string& protocol\_name, int name\_width,      int cycles\_width)  {      const std::string separator = std::string(name\_width, '-') + "|"          + std::string(cycles\_width, '-');      std::cout << center(protocol\_name, name\_width + cycles\_width + 1) << "\n";      std::cout << separator << "\n";      std::cout << center("Name", name\_width) << "|"                << center("Cycles", cycles\_width) << "\n";      std::cout << separator << "\n";  }  void print\_performance\_row(const std::string& name, unsigned int cycles,      int name\_width, int cycles\_width)  {      std::cout << left(name, name\_width) << "|" << prd(cycles, cycles\_width)                << "\n";  }  void print\_performance\_table(      const std::string& protocol\_name,      const std::vector<std::tuple<std::string, unsigned int>>& performance\_data,      int name\_width, int cycles\_width)  {      print\_protocol\_header(protocol\_name, name\_width, cycles\_width);      for (const auto& entry : performance\_data) {          print\_performance\_row(std::get<0>(entry), std::get<1>(entry),              name\_width, cycles\_width);      }  } |